A Generic Describing Method of Memory Latency Hiding in a High-level Synthesis Technology

Akira Yamawaki and Seiichi Serikawa
Kyushu Institute of Technology, Kitakyushu, Japan
Email: {yama, serikawa}@ecs.kyutech.ac.jp

***Abstract*—We show a generic describing method of hardware including a memory access controller in a C-based high-level synthesis technology (Handel-C). In this method, a prefetching mechanism to improve the performance by hiding memory access latency can be systematically described in C language. We demonstrate that the proposed method is very simple and easy through a case study. The experimental result shows that although the proposed method introduces a little hardware overhead, it can improve the performance significantly.**

***Index Terms*—high-level synthesis, memory access, data prefetch, FPGA, Handel-C, latency hiding**

I. Introduction

The system-on-chip (SoC) used for the embedded products must achieve the low cost and respond to the short life-cycle. Thus, to reduce the development burdens such as the development period and cost for the SoC, the high-level synthesis (HLS) technologies for the hardware design converting the high abstract algorithm-level description like C, C++, Jave, and MATLAB to a register transfer-level (RTL) description in a hardware description language (HDL) have been proposed and developed. In addition, many researchers in this research domain have noticed that the memory access latency has to be hidden to extract the inherent performance of the hardware.

Some researchers and developers have proposed the hardware platforms combining the HLS technology with an efficient memory access controller (MAC). The MACs as a part of the platforms at high-level description such as C, C++ and MATLAB have been shown [1, 2, 3]. In these platforms, the designer has only to write the data processing hardware with the simple interfaces communicating with the MACs in order to access to the memory. However they did not consider hiding the memory access latency.

Ref. [4] describes some memory access schedulers built as hardware to reduce the memory access latency by issuing the memory commands efficiently and hiding the refresh cycle of the DRAM memory. This proposal hides only the latencies native to the DRAM such as the RAS-CAS latency, the bank switching latency and the refresh cycle. Thus, this scheduler cannot hide the application-specific latency like the streaming data buffering, the block data buffering, and the window buffering. Some HLS tools [5, 6, 7, and 8] can describe the memory access behavior in C language as well as the data processing hardware. Ref. [5, 6, 7] however have never shown a generic describing method to hide memory access latency. Thus, the designers must have a deep knowledge of the HLS tool used and write the MAC well relying on their skill. Ref. [8] needs the load/store unit described in HDL which is a tiny processor dedicated to memory access. Thus, the high-speed simulation at C level cannot be performed in an early stage during the development period. We propose a generic method to describe the hardware including a functionality hiding application-specific memory access latency at C language. This paper pays attention to the Handel-C [9] that is one of the HLS tools that has been widely used for the hardware design.

To hide memory access latency, our method employs the software-pipelining [10] which is widely used in the research domain of the high performance computing. The software-pipelining reconstructs the programming list by copying the load and store operations in the loop into front of the loop and into back of the loop respectively. Since this method is very simple, the user can easily use it and describe the hardware with the feature hiding memory access latency. Consequently, the generic method of the C-level memory latency hiding can be introduced into the conventional HLS technology.

Generally, the performance estimation is performed to estimate the effect of the software pipelining. The conventional method [10] uses the average memory latency for the processor with a write-back cache memory. For a hardware module in an embedded SoC, such cache memory is very expensive and cannot be employed. Thus, new performance estimation method is needed. Thus, we propose the new estimation method considering of the hardware module to be mounted onto the SoC. The rest of the paper is organized as follows. Section 2 shows the target hardware architecture. Section 3 describes the load/store functions in Handel-C, Section 4 describes the templates of the memory access and data processing. Section 5 demonstrates the software-pipelining method to hide memory access latency. Section 6 explains new method of the performance estimation based on the hardware architecture shown in Section 2, considering the load and store for the software-pipelining. Section 7 shows the experimental results. Finally, Section 8 concludes this paper and remarks the future work.

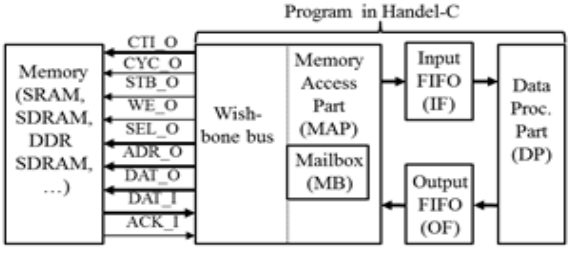


Figure.1 Target architecture.

© 2012 ACEEE
DOI: 01.IJIT.02.02.44

ACEEE

II. TARGET HARDWARE ARCHITECTURE

Fig. 1 shows the architecture of the target hardware. This architecture is familiar to the hardware designers and the HLS technologies. This architecture consists of the memory access part (MAP), the input/output FIFOs (IF/OF) and the data processing part (DP). They are written in C language of the Handel-C. The MAP is a control intensive hardware to load and store the data in the memory. The MAP accesses to the memory via wishbone bus [11] in a conventional request-ready fashion. The MAP has the register file including the control/status registers, mailbox (MB), of the hardware module. The designer of the hardware module can arbitrarily define each register in the MB. Each register can be accessed by the external devices such as the embedded processor. The DP is a data processing hardware which processes the streaming data. Any HLS technology is good at handling the streaming hardware.

The MAP accesses the memory according to the memory access pattern written in C. The MAP loads the data into the IF, converting the memory data to the streaming data. The DP processes the streaming data in the IF and stores the processed data into the OF as streaming data. The MAP reads the streaming data in the OF and stores the read data into the memory according to the memory access pattern. The MAP and the DP are decoupled by the input/output FIFOs. Thus, the hardware description is not confused about the memory access and the data processing. Generally, any HLS technology has the primitives of the FIFOs.

III. LOAD/STORE FUNCTION

Generally, the memory such as SRAM, SDRAM and DDR SDRAM support the burst transfer which includes the continuous 4 to 8 words. Thus, we describe the load/store primitives supporting burst transfer as function calls as shown in Fig. 2 and Fig. 3 respectively.

As for the load function shown in Fig. 2, the bus request is issued setting WE_O to 0 in the lines 2-5. When the acknowledgement (ACK_I) is asserted by the memory, this function performs the burst transfer of loading in the lines 6-20. In the Handel-C, “par” performs the statements in the block in parallel at 1 clock. In contrast, the “seq” performs the statements in the block sequentially. Each statement consumes 1 clock. That is, the continuous words in the burst transfer are pushed into the input FIFO (IF) one by one. The ‘!’ is the primitive pushing into the FIFO in Handel-C. When the specified FIFO (in this example it is IF) is full, this statement is blocked until the FIFO has an empty space. When burst transfer finishes, the current address is added by the burst length as shown in the line 17 in order to issue the next burst transfer of loading. As for the store function shown in Fig. 3, this function attempts to pop the output FIFO (OF) to store the data processed by the data processing part (DP) in the line 2. The ‘?’ is the primitive popping the FIFO in the Handel-C. When the OF is empty, this statement is blocked until the DP finishes the data processing and pushes the result into the OF.

```
1:void mem_load (uint WORD_WIDTH *addr){
2: do{ //Bus Request.
3:  par{CTI_O=0x2;CYC_O=1;STB_O=1;WE_O=0;
4:       SEL_O=0xf;ADR_O=*addr;}
5: }while(ACK_I==0);
6: seq(i=0;i<BURST_LEN;i++){
7:  par{
8:   IF ! DAT_I; //Push input data to IF.
9:   if(i==(BURST_LEN-2))
10:    CIT_O=0x7; //Inform burst end.
11:  //Terminate burst transfer.
12:  if(i==(BURST_LEN-1))
13:   par{
14:    CIT_O=0;CYC_O=0;STB_O=0;WE_O=0;
15:    SEL_O=0;ADR_O=0;
16:    //Update load address.
17:    *addr+=BURST_LEN*WORD_SIZE;
18:   }
19:  }
20: }
21:}
```

Figure.2 Load function.

```
1:void mem_store (uint WORD_WIDTH *addr){
2: OF ? temp; //Pop OF to temp.
3: do{ //Bus Request.
4:  par{CTI_O=0x2;CYC_O=1;STB_O=1;WE_O=1;
5:       SEL_O=0xf;ADR_O=*addr;}
6: }while(ACK_I==0);
7: DAT_O=temp; //Output first word.
8: seq(i=1;i<BURST_LEN;i++){
9:  par{
10:  OF ? DAT_O; //Pop OF to bus output.
11:  if(i==(BURST_LEN-2))
12:    CIT_O=0x7; //Inform burst end.
13:  //Terminate burst transfer.
14:  if(i==(BURST_LEN-1))
15:   par{
16:    CIT_O=0;CYC_O=0;STB_O=0;WE_O=0;
17:    SEL_O=0;ADR_O=0;
18:    //Update load address.
19:    *addr+=BURST_LEN*WORD_SIZE;
20:   }
21:  }
22: }
23:}
```

Figure.3 Store function

Then, the bus request is issued setting WE_O to 1 in the lines 3-6. When the acknowledgement (ACK_I) is asserted by the memory, this function performs the burst transfer of storing in the lines 7-22. During the burst transfer, the OF is popped and the popped data is outputted into the output port (DAT_O) one by one per 1 clock. As similar to the load function, the current address is added by the burst length for the next transfer in the line 19.

```
1:void MAP (void){
2: while(1){
3:  //Waiting for invocation.
4:  while(MB[0]==0) delay;
5:  par{
6:   MB[0]=0; //Reset start flag.
7:   MB[4]=0; //Reset end flag.
8:   read_addr =MB[1]; //Get read address.
9:   write_addr=MB[2]; //Get write address.
10:  end_addr  =MB[3]; //Get end address.
11:  }
12:  while( read_addr < end_addr ){
14:   mem_load (&read_addr ); //Mem=>IF.
16:   mem_store(&write_addr); //OF =>Mem.
15:  }
16:  MB[4] = 1; //Set end flag.
17: }
18:}
```

Figure. 4 Template of memory access part

© 2012 ACEEE
DOI: 01.IJIT.02.02.44

```
1:void DP(void){
2: uint WORD_WIDTH i_dat[BURST_LEN];
3: uint WORD_WIDTH o_dat[BURST_LEN];
4:
5: while(1){
6:  //Pop from IF into i_dat.
7:  seq(i=0;i<BURST_LEN;i++){IF ? i_dat[i];}
8:   /* Data processing using i_dat.  */
9:   /* Results are stored into o_dat.*/
10:  //Push the processed data into OF.
11:  seq(i=0;i<BURST_LEN;i++){OF ! o_dat[i];}
12: }
13:}
```

Figure. 5 Template of data processing part

```
1:void main(void){
2: par{
3:  MB ( ); //Behavior of MB as bus slave.
4:  MAP( ); //Memory access part.
5:  DP ( ); //Data processing part.
6: }
7:}
```

Figure. 6 Whole of hardware module

IV. MEMORY ACCESS AND DATA PROCESSING

By using load/store functions shown in Fig. 2 and Fig. 3, the memory access part can be written easily as shown in Fig. 4. Since Fig. 4 is a template of the memory access part (MAP), it can be modified as the designer's like. The MAP cooperates with the data processing part (DP) shown in Fig. 5. Fig. 5 is also the template. So, the designer must describe the data processing in the lines 8 to 10.

In this example, the mailbox #0 (MB[0]) is used for the start flag of the hardware module. The MB[4] is used for the end flag indicating that the hardware module finishes the data processing completely. The MB[1], MB[2] and MB[3] are used for the parameters of the addresses used. By utilizing the mailbox (MB), different memory access patterns can be realized flexibly.

When the MAP is invoked, it loads the memory by burst transfer and pushes the loaded words into the input FIFO (IF) in the line 14. The DP is blocked by the pop statement to the IF in the line 7. When the MAP pushes the IF, the DP pops the words in IF to the temporary array (i_dat[i]). Then, the DP processes the popped data and generates the result into the temporary array (o_dat[i]). At the same time, the

```
1:void MAP (void){
   …
12:  mem_load(&read_addr); //prologue
13:  while( read_addr < end_addr ){ //kernel
14:   mem_load (&read_addr ); //Mem=>IF.
15:   mem_store(&write_addr); //OF =>Mem.
16:  }
17:  mem_store(&write_addr); //epilogue
18:  MB[4] = 1; //Set end flag.
19: }
20:}
```

Figure. 7 Software pipelining

MAP is blocked by the pop statement in the line 16 until the DP pushes the result data into the OF. When the DP pushes the processed data into the OF, the MAP can perform the burst transfer of storing.

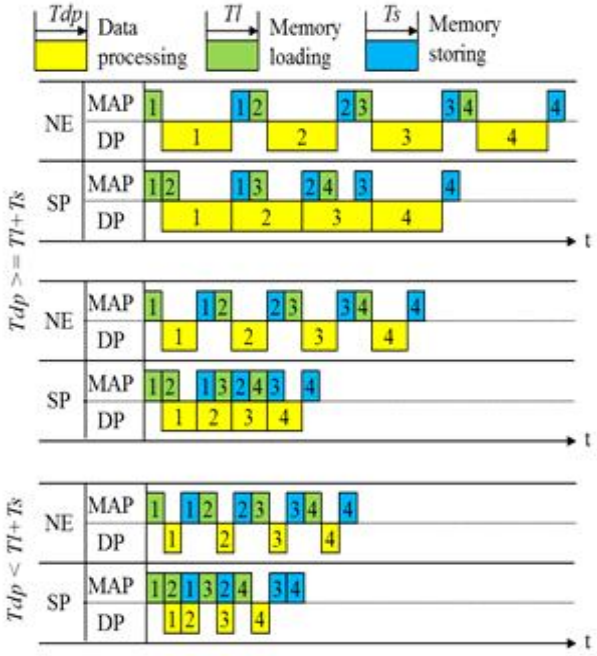


Figure. 8 Execution snapshot

Until all data are processed, above flow is repeated. When the data processing finishes completely, the MAP exits the main loop in the lines 12 to 15 and sets the end flag to 1 in the line 16. Consequently, the MAP is blocked in the line 4 and the DP is blocked in the line 7. Thus, this hardware module can be executed with new data.

Fig. 6 shows how to describe the whole of the hardware module. The MB ( ) is the function of the behavior of the mailbox as bus slave. Due to paper limitation, its detail is omitted. In this program, the MB, the MAP and the DP are executed in parallel.

V. SOFTWARE PIPELINING

To hide the memory access latency, the software pipelining has been applied to the original source code of the application program [10]. Our proposal applies the software pipelining to the program of the memory access part (MAP) as shown in Fig. 4. Fig. 7 shows the overview of the software pipelining to the MAP program.

In the software pipelining, the burst transfer of loading (mem_load) and storing (mem_store) in the main loop are copied to the front of the main loop and the back of it respectively. In the main loop, the data used at the next iteration is loaded at the current iteration. Thus, the memory accesses of the MAP are overlapped with the data processing part (DP). In addition, the size of the input and output FIFOs is doubled.

VI. PERFORMANCE ESTIMATION

In the conventional software pipelining, its effect is estimated by using the average memory latency [10]. However the average memory latency is measured on the processor with a cache memory. So, it is not applied to the hardware module used in SoCs. The hardware module generally does not have a cache memory. So, the memory access latency

© 2012 ACEEE
DOI: 01.IJIT.02.02.44

due to every load and store should be considered. When *Tdp* is the data processing time of the data processing part (DP), the normal execution time (*Tne*) without software pipelining can be calculated as following expression.

$$Tne = (Tl + Tdp + Ts) \times n. \quad (1)$$

where *Tl* is the load latency, *Ts* is the store latency and *n* is the number of iterations. The execution snapshot when the software pipelining is applied is shown as Fig. 8. The NE means the normal execution and the SP means the software pipelined execution. The number of each square indicates the iteration number.

As shown in upper side of Fig. 8, if *Tdp* is greater than equal to *Tl*+*Ts*, the memory access latency is enough hidden. So, the data processing time is dominant of the total execution time. In contrast, as shown in lower side of Fig. 8, if *Tdp* is lower than *Tl* + *Ts*, the memory latency affects the performance. The total execution time comes closer to the memory bottleneck. The software pipelined execution time (*Tsp*) can be calculated as follows.

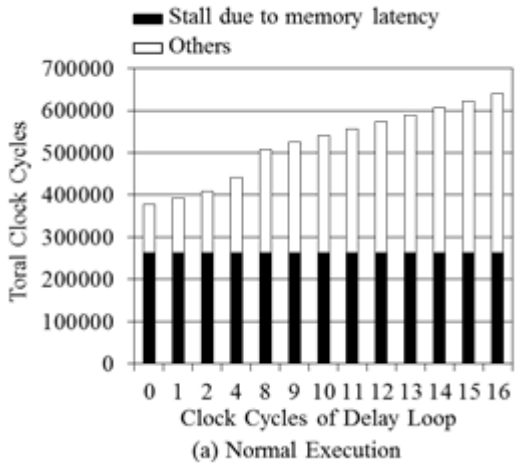
$$Tsp = \begin{cases} Tdp \times n + Tl + Ts, Tdp \geq (Tl + Ts). \\ Tdp + (Tl + Ts) \times n, Tdp < (Tl + Ts). \end{cases} \quad (2)$$

When the speedup ratio (*Tne* / *Tsp*) is greater than 1, the software pipelining is effective for the hardware module.

VII. EXPERIMENT AND DISCUSSION

A. Performance Evaluation

In order to confirm the effect of the software pipelining to the performance, we have described the whole of hardware shown in Fig. 4, Fig. 5, Fig. 6 and Fig. 7 in Handel-C (DK Design tool 5.4 of Mentor Graphics). For the data processing part in Fig. 5, we insert the delay loop into the lines 8 to 10 as a data processing. Varying the number of clock cycles of the delay loop, we have measured the execution time by using the logic simulator (Modelsim10.1 of Mentor Graphics). In addition, we have assumed that a 32bits width a DDR SDRAM with the burst length of 4 is used. Its load latency is 8 clock cycles and its store latency is 9 clock cycles. The number of iterations has been set to 16384. That is, the data size was 256KB. The clock frequency has been set to 100MHz.



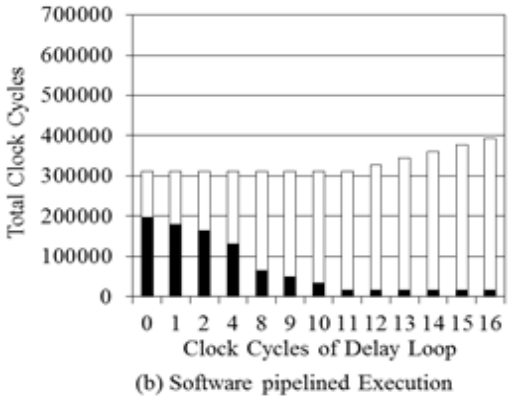


Figure. 9 Performance evaluation

The experimental result is shown in Fig. 9 which is the breakdown of the execution time of the data processing part (DP). The horizontal axis means the number of clock cycles of the delay loop. The vertical axis shows the number of clock cycles consumed until the hardware finishes the execution for all data. The black bar is the stall time of the DP due to the memory latency. The white bar is the clock cycles consumed

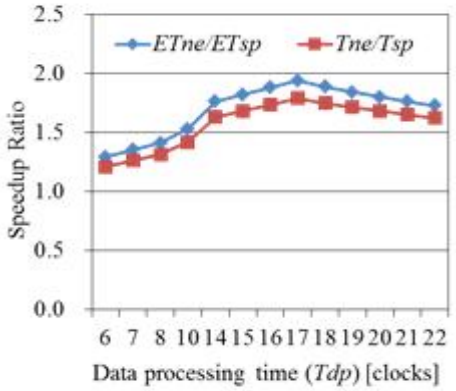


Figure. 10 Speedup ratio

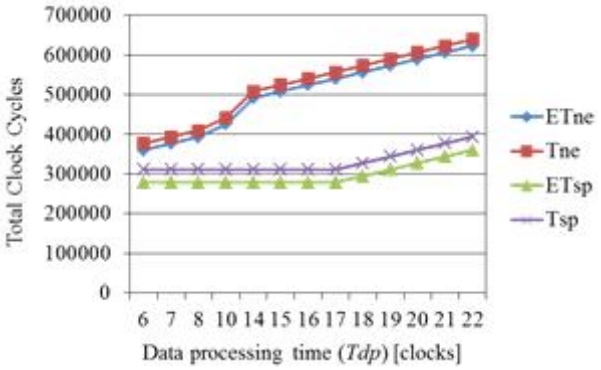


Figure. 11 Total Execution time

by the other execution except for the memory stall time. Although the clock cycle of the delay loop is zero, the DP needs some clock cycles in addition to the stall time. This is because the overhead pushing and popping FIFOs is included. This overhead is 6 clock cycles per iteration. The hiding memory latency by software pipelining can improve the performance significantly compared with the normal execution.

Fig. 10 shows the speedup ratio. The *ETne* and the *ETsp* mean the calculated normal execution time and the software-

© 2012 ACEEE
DOI: 01.IJIT.02.02.44

pipelined execution time respectively. The *Tne* and the *Tsp* are measured results. By hiding memory latency, the speed-ups of 1.21 to 1.79 can be achieved. In addition, the result shows that our estimation method can get the same tendency to the measured results.

Fig. 11 shows the results of the estimated execution time and the measured execution time. Where the data processing time (*Tdp*) is lower than *Tl + Ts =17*, the performance is closer to the memory bottleneck plus the inherent overhead due to FIFO access, by overlapping the data processing onto the memory access. Once the *Tdp* becomes larger than equal to the *Tl + Ts=17*, the data processing occupies the performance. Thus, the execution time is increasing as the delay loop (computation) is becoming larger.

B. Hardware Size

By reconstructing the hardware description as shown in Fig. 7 for applying the software pipelining, the hardware size may increase compared with the normal version. To confirm this hardware overhead, we have implemented the Handel-C description into the FPGA. The target FPGA is Spartan6 and the ISE13.1 is used for implementation. The result shows that the software pipelined version uses 1.09 times of logic resources than the normal version. The difference between clock frequencies of both versions is about 2% only. Thus, the hardware overhead due to applying the software pipelining is very small and it can be compensated by performance improvement.

VIII. CONCLUSION

We have shown a generic describing method of hardware including a memory access controller in a C-based high-level synthesis technology, Handel-C. This method is very simple and easy, so any designer can employ the memory hiding method for the design entry in C language level. The experimental result shows that the proposed method can improve the performance significantly by hiding memory access latency. The new performance estimation can be useful because the estimated performances have shown the same tendency to the measured results. The proposed method does not introduce the significant bad effect to the normal version hardware. As future work, we plan to apply our method to other commercial HLS tools. Also, we will use more application programs and practically integrate hardware modules into a SoC.

ACKNOWLEDGMENT

This research was partially supported by the Grant-in-Aid for Young Scientists (B) (22700055).

REFERENCES

[1] T. Papenfuss and H. Michel, "A Platform for High Level Synthesis of Memory-intensive Image Processing Algorithms," in *Proc. of the 19th ACM/SIGDA Int'l Symp. on Field Programmable Gate Arrays*, pp. 75–78, 2011.

[2] H. Gadke-Lutjens, and B. Thielmann and A. Koch, "A Flexible Compute and Memory Infrastructure for High-Level Language to Hardware Compilation," in *Proc. of the 2010 Int'l Conf. on Field Programmable Logic and Applications*, pp. 475–482, 2010.

[3] J. S. Kim, L. Deng, P. Mangalagiri, K. Irick, K. Sobti, M. Kandemir, V. Narayanan, C. Chakrabarti, N. Pitsianis, and X. Sun, "An Automated Framework for Accelerating Numerical Algorithms on Reconfigurable Platforms Using Algorithmic/Architectural Optimization," in *IEEE Transactions on Computers*, vol. 58, No. 12, pp. 1654–1667, December, 2009.

[4] A. M. Kulkarni and V. Arunachalam, "FPGA Implementation & Comparison of Current Trends in Memory Scheduler for Multimedia Application," in *Proc. of the Int'l Workshop on Emerging Trends in Technology*, pp.1214–1218, 2011.

[5] Mitrionics, *Mitrion User's Guide 1.5.0-001*, Mitrionics, 2008.

[6] D. Pellerin and S. Thibault, *Practical FPGA Programming in C*, Prentice Hall, 2005.

[7] D. Lau, O. Pritchard and P. Molson, "Automated Generation of Hardware Accelerators with Direct Memory Access from ANSI/ ISO Standard C Functions," *IEEE Symp. on Field-Programmable Custom Computing Machines*, pp.45–56, 2006.

[8] A. Yamawaki and M. Iwane, "High-level Synthesis Method Using Semi-programmable Hardware for C Program with Memory Access," *Engineering Letters*, Vol. 19, Issue 1, pp. 50–56, 2011.

[9] Mentor Graphics, "Handel-C Synthesis Methodology," *http://www.mentor.com/products/fpga/handel-c/*, 2011.

[10] S. P. Vanderwiel, "Data Prefetch Mechanisms," *ACM Computing Surveys*, Vol. 32, No. 2, pp. 174–199, 2000.

[11] OpenCores, *Wishbone B4 WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*, OpenCores, 2010.

© 2012 ACEEE
DOI: 01.IJIT.02.02.44